HW 8
"Hello World" Linux Kernel Module

EELE 467
Due date: 11/08/2024

The goal of this homework is to introduce some basic concepts about Linux kernel modules and learn how to compile them. You will create a "Hello World" kernel module and use `dmesg` and `grep`, two common Unix utilities, to find debug messages you printed from your kernel module.

You will do all of this assignment in the Ubuntu virtual machine and on the SoC (i.e., connected to the ARM processor using PuTTY).

This assignment is very similar to lab 9 in the textbook. Sections 9.2 and 9.3 in the textbook contain some additional information which may be helpful.

Introduction and Background

Before we can write a "Hello World" kernel module, we first need to learn a bit about kernels and kernel modules.

The kernel is at the core of any operating system. It is responsible for things such as resource management, process scheduling, filesystems, and hardware control, just to name a few. It is the layer that sits between user processes and hardware. The Linux kernel is a monolithic kernel used in Linux (or GNU/Linux, if you prefer) operating systems.

We will be writing Linux kernel *modules*, which are pieces of code that can be added to the kernel at runtime. Although there are multiple types of modules, we will be focused on *device drivers*—code that interfaces with a hardware device. The terms "kernel module" and "device driver" are sometimes used interchangeably (and will likely be interchanged in this course), although there are differences.

In embedded systems, it is common to compile device drivers and other functionality directly into the kernel. In either case, the module/driver code is the same; the primary difference is in how the code is *linked*: when compiled into the kernel, the driver is *statically* linked, as opposed to *dynamically* linked in the case of loadable kernel modules.

As we will be diving into kernel territory, we will make the distinction between *kernel space* and *user space*. *Kernel space* is where kernel code lives and runs, whereas *user space* is where normal applications run. Beyond the obvious statement of which programs run where, there are important differences between kernel space and user space:

- kernel space has its own memory space, which is not accessible from user space
- kernel code executes at the highest CPU privilege level[1], whereas user programs execute at the lowest CPU privilege level

[1] CPUs have multiple privilege levels, which are used to enforce protection rings. In short, privilege levels place restrictions on what operations/instructions processes have access to.

Device drivers can be, and sometimes are, written in user space. In fact, we created a user space driver when we controlled our LED state machine by accessing `/dev/mem` with `mmap()`. While user space drivers occasionally have advantages, most drivers are written in kernel space, which is what we will be focusing on.

Kernel Modules

With some preliminary terminology out of the way, we can dive into the basics of kernel modules. As stated before, kernel modules are code that gets dynamically linked with the kernel at runtime. There are of course a lot of details behind the scenes to make dynamic loading/unloading work, though we won't get into most of them here. We will focus on what our module code needs to support loading/unloading, as well as the *user space* programs we will use to load/unload modules.

Differences from User Space

We already know that kernel space and user space are different, but there are some *programming* differences we need to be aware of as well:

- The kernel doesn't have access to the C standard library, thus you can't use any of those functions. However, the kernel reimplements some standard library functions. For example, `printk()` is the kernel equivalent of `printf()`. Look through the Linux Kernel API, especially the "Basic C Library Functions" section, to see what is available.
- Floating point arithmetic isn't allowed; this is largely because of the overhead of having to save and restore the floating point unit's state on every transition between user space and kernel space.
- The kernel's stack is very small. Consequently, you should dynamically allocate any large data structures.

Initialization and Cleanup Functions

Nearly all kernel modules export callback functions for initialization and cleanup, which the kernel will call when loading and unloading the module, respectively. Modules without a cleanup function can't be unloaded. Modules can be loaded without an initialization function, but any non-trivial device driver will have to do some sort of initialization (setting default register values, enabling the device, registering sysfs entries, etc.).

Macros and definitions for initialization and cleanup are found in `<linux/init.h>` (you can browse the Linux source code, including `init.h`, via https://elixir.bootlin.com).

The following listings show what the init and cleanup function declarations look like (the actual function names can be whatever you want).

```
static int __init init_function(void)
```

Listing 1: Example init function declaration

```
static void __exit cleanup_function(void)
```

Listing 2: Example cleanup function declaration

The `__init` and `__exit` tokens are used to tell the kernel that the functions are used only during initialization and cleanup, respectively. This allows the kernel to drop the initialization function after the module is loaded, thereby freeing up memory. If you build your module directly into the kernel, `__exit` tells the kernel to discard the cleanup function, as the module can't be removed from the kernel. `__initdata` is another useful token, which specifies that the corresponding data structure can be dropped after initialization. See init.h for details.

Another thing to note is that the functions should be declared **`static`** because they are not meant to seen outside the module's file. More concretely, **`static`** functions are scoped to the file in which they are defined, meaning other functions can't link to and use them. There is, however, a caveat here, because the kernel *has* to call a module's init and exit functions. This is accomplished by the `module_init` and `module_exit` macros, which are defined in `<linux/module.h>`. Essentially, these macros export function pointers to your init and cleanup functions, such that the kernel has access to the function pointers and can thus call the functions. Listing 3 shows how to use the `module_init` and `module_exit` macros. In summary, declaring your init and cleanup functions as **`static`** ensures that only the kernel can call those functions.

```
module_init(init_function);
module_exit(cleanup_function);
```

Listing 3: Exporting init and cleanup callbacks

`module_init` and `module_exit` **must** be used, otherwise the kernel will never call your init and cleanup functions.
Basic API documentation for `module_init` and `module_exit` is found in the `<linux/module.h>` and in the Driver Basics API documentation.

`MODULE` Macros

Various metadata macros are defined in `<linux/module.h>`. In particular:

- `MODULE_AUTHOR()`: who wrote the module
- `MODULE_DESCRIPTION()`: description of what the module does
- `MODULE_VERSION()`: code version number
- `MODULE_LICENSE()`: specifies which license applies to your code. If a preferred free license is not used, the module is assumed to be proprietary, which will taint the kernel; we don't need to be particularly worried about these issues in this class as proprietary kernel modules can be loaded, but your code has to be GPL-licensed if you want it to be merged into the mainline kernel. If you want a more permissive license, you can dual-license your module, e.g. `MODULE_LICENSE("Dual MIT/GPL")`. See the kernel license-rules for more info on the accepted `MODULE_LICENSE()` strings.

By convention, these macros are placed at the end of the module's source file. In the least, you will be **required to use `MODULE_AUTHOR`** in this class for all of your modules.

See module.h for more info.

Compiling Kernel Modules

Kernel modules must be built with the kernel's build system, kbuild. Modules can either be built in-tree (within the kernel's source tree) or out-of-tree (externally). We will be building our modules out-of-tree.

To build external modules, we need a copy of the kernel configuration and header files for the kernel we are building for. Right now, we will use our Linux distribution's kernel headers package for this; later on, we will build the kernel ourselves.

> Kernel modules must be compiled against the kernel version you are using; doing otherwise will result in the module not loading.

On Ubuntu, the kernel headers packages are of the form `linux-headers-<kernel-release>`. Fortunately, Ubuntu has a metapackage, `linux-headers-generic`, that will resolve dependencies and install the correct headers for the kernel you are running. Not all distributions will provide a metapackage for kernel headers. In those cases, you will have to determine what kernel release you are running; this can be done with `uname` (see the uname man page to figure out how to return your kernel release: `man uname`).

kbuild uses a special build file syntax, described in the Linux Kernel Makefiles documentation. All we need to be concerned with right now is what are known as "goal definitions". For building a module that only has a single source file, our goal definition will look like this: `obj-m := <module-name>.o`, where `<module-name>` is the name of your source file (without an extension).

If we were building in-tree, all we would need is that single goal definition. But since we are building out of tree, we need to tell `make` where the kernel source is and where our module is:

```
make -C <kernel-source-path> M=<module-path>
```

Rather than typing this command out all the time, kernel developers create a Makefile like shown in Listing 4.

```
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
obj-m  := hello.o

else
# normal makefile

# path to kernel directory
KDIR ?= /lib/modules/`uname -r`/build

default:
    $(MAKE) -C $(KDIR) M=$$PWD

clean:
    $(MAKE) -C $(KDIR) M=$$PWD clean
endif
```

Listing 4: Example Makefile for building an external module

The Makefile in Listing 4 is invoked by just running `make`. When you are out-of-tree, the Makefile will call the appropriate `make` command to build the module using kbuild, which will make another pass through the Makefile, this time using the kbuild part of the Makefile.

After compiling a module, a `<module-name>.ko` file will be created; this is the kernel module artifact that we will load/unload.

For more information on building external modules, refer to this section of the Kbuild documentation.

Loading/Unloading Kernel Modules

We will use `insmod` to insert kernel modules, and `rmmod` to remove kernel modules. `modprobe` is a more robust program, but it only looks for modules in `/lib/modules/`uname -r``, which is not necessarily where our modules will be; we could use `modprobe`, but we will opt for the simpler `insmod`/`rmmod`.

Create a "Hello World" Kernel Module

Using the information you learned above, create a "Hello World" kernel module that prints "Hello, world" when loaded and prints "Goodbye, cruel world" when unloaded.

- Put your name as the module author.
- Put your source code and Makefile in `linux/hello-world`.

> You will want to look at the printk basics documentation to learn how to use `printk()`.

> Create initialization and cleanup functions.

Compiling "Hello World"

You can use the Makefile given in Listing 4 to compile your module (you will have to change the object name in the Makefile if you don't name your module `hello.c`).

Before compiling, you will need to install the Linux kernel headers:

```
sudo apt install linux-headers-generic
```

Verification

Load and unload your kernel module using `insmod` and `rmmod`.

Verify that your kernel module is working by finding your printk statements in the kernel's ring buffer by using `dmesg` and `grep`. `dmesg` is used to print or control the kernel's ring buffer, and `grep` is a utility for finding string patterns. You will need to use pipes, which will allow you to use `grep` on the output of `dmesg`.

Verify that you are the author of the kernel module by running `modinfo <your-module-name.ko>`

Cross-compiling your kernel module for the ARM processor

Now that we've successfully compiled and loaded a basic kernel module, we're going to cross-compile the kernel module. All the actual device drivers we write will run on the SoC FPGA, so we need to learn how to cross-compile kernel modules.

Prerequisites

1. If you haven't already, install the Linaro armhf cross-toolchain:

   ```
   sudo apt install gcc-arm-linux-gnueabihf
   ```

   `gnueabihf` stands for GNU embedded-application binary interface hard float

2. Install packages required for kernel compilation; in particular:
   - build-essential
   - bison
   - flex

Cross-compiling the kernel

In the previous section, we used Ubuntu's kernel headers package to provide the files we needed to compile our kernel module. This time, we will use the full kernel source tree; there are several reasons for doing this:

- Your Linux distro might not provide a kernel headers package for the armhf architecture (if it does, installing packages for multiple architectures can get confusing).
- We want to ensure that we are cross-compiling our modules for the correct kernel version; the easiest way to do this is to build both the kernel and the module ourselves.

We'll start by downloading and building the Linux kernel:

1. Download Intel's SoC FPGA kernel repo:

   ```
   git clone --depth 100 https://github.com/altera-opensource/linux-socfpga
   ```

   NOTE: `--depth 100` means we're only downloading the last 100 commits. The Linux kernel has well over one million commits, which takes a long a time to clone...

2. Navigate to the `linux-socfpga` directory (the git repo you just downloaded)

3. Configure your kernel with the default socfpga configuration options:

   ```
   make ARCH=arm socfpga_defconfig
   ```

   `ARCH=arm` is an environment variable that tells `make` we are cross-compiling for arm. `socfpga_defconfig` contains all of the default Kconfig values for socfpga, as defined by Intel and the community; this file is located in `linux-socfpga/arch/arm/configs/`.

4. Compile the kernel:

   ```
   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j$(nproc)
   ```

The CROSS_COMPILE environment variable specifies the cross-toolchain's prefix. The -j flag specifies the number of simultaneous jobs, which we set to the number of logical processors on your machine; this results in faster compile times (yay parallel processing!).

Compiling the kernel can take a long time. On an i9-9900k (16 logical processors) with 32 GB of RAM, compilation took 1.5 minutes. It will likely take 10+ minutes on your virtual machine.

After compilation, the kernel image is located at `linux-socfpga/arch/arm/boot/zImage`. The zImage is a self-extracting, compressed version of the kernel image that gets loaded by the U-Boot bootloader. Since we are having U-Boot download the zImage over tftp, you need to copy the new zImage file to your tftp directory: `/srv/tftp/de10nano/kernel/`.

You may recognize the ARCH and CROSS_COMPILE environment variables. If you like, you can source the `arm-env.sh` file to export these variables for you, instead of typing them with the `make` command.

Cross-compiling the "Hello World" module

Since you already created your "Hello World" module, all we need to do is modify our Makefile to cross-compile the module. Listing 5 highlights the lines that you need to modify.

```
1  ifneq ($(KERNELRELEASE),)
2  # kbuild part of makefile
3  obj-m  := hello.o
4
5  else
6  # normal makefile
7
8  # path to kernel directory
9  KDIR ?= ../../linux-socfpga
10
11 default:
12     $(MAKE) -C $(KDIR) ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- M=$$PWD
13
14 clean:
15     $(MAKE) -C $(KDIR) M=$$PWD clean
16 endif
```

Listing 5: Makefile for cross-compilation. KDIR ?= needs to point to the directory where you put the kernel. The ARCH and CROSS_COMPILE environment variables are included in the Makefile for convenience, though they can be exported externally before running `make` if preferred.

Once you have modified your Makefile, you can build the kernel module as before. To verify that your module was built for the ARM architecture, you can use `modinfo` to look at the module's `vermagic` string, or use the `file` command.

Move the compiled module (.ko file) to your `/srv/nfs/de10nano/ubuntu-rootfs/home/soc/` in your armhf rootfs.

Verification

Log into your SoC via PuTTY, then load and unload your kernel module using `insmod` and `rmmod`.

In PuTTY, verify that your kernel module is working by finding your printk statements in the kernel's ring buffer by using `dmesg` and `grep`.

Deliverables

Submit your HW 8 assignment using the workflow that's detailed at `docs/workflow.md` and the submission template at `docs/submission-template.md`.

Make sure your kernel module source code and Makefile are in located in `linux/hello-world/`.

In your `hw8.md` file, include screenshots of the following on both your Ubuntu VM and on the SoC.

- Show "Hello world" and "Goodbye, cruel world" being printed in the kernel's log buffer.
- Use `uname` to show all of your system's information (consult `uname`'s `man` page to figure out how to do this).
- Show the output of `modinfo` on your kernel module.